ERDC MSRC/PET TR/00-33

Comparison of OpenMP and Pthreads within a Coastal Ocean Circulation Model Code

by

Clay P. Breshears
Phu Luong

13 July 2000

Abstract

Numerical grid generation techniques play an important role in the numerical solution of partial differential equations on arbitrarily shaped regions. For coastal ocean modeling, a one-block grid covering the region of interest, while commonly used, has many disadvantages, including large numbers of unused grid points around bodies of water with complicated coastlines, large memory requirements, and poor resolution for a large body of water. The Multiblock Grid Princeton Ocean Model (MGPOM) ocean circulation code is used to overcome these problems.

The MGPOM code uses multiple grid blocks designed to best fit the region of interest. Computation within each individual grid block can be done in parallel using MPI to synchronize processing and communicate shared data. Since not all grid blocks are of the same size, the work load varies between MPI processes. Such load imbalances force processors with small grid blocks to remain idle at communication points. To alleviate this imbalance, we investigate the use of threads within the parallel computations.

In this study, we compare the two threading models of OpenMP and Pthreads with an eye toward correcting the load imbalance between blocks. Along with overall execution time performance enhancements on the MGPOM code achieved by use of each model, the ease of programming with each model will also be examined. By keeping the focus at a high level and the use of the MGPOM production code as a test case, it is intended that users will gain a better insight into the strengths and pitfalls of each model.

Comparison of OpenMP and Pthreads within a Coastal Ocean Circulation Model Code

Clay P. Breshears[1] Phu Luong[2]

July 13, 2000

[1]Rice University; PET On-site Scalable Parallel Programming Tools Lead
[2]University of Texas, Austin; PET On-site Environmental Quality Modeling Lead

1 Introduction

Over the years, the traditional one-block rectangular grid has been used for ocean circulation modeling. This technology encounters difficulty on computational grids with high resolution owing to the large memory and computing requirements. For a large body of water, such as an ocean with complicated coastlines, the number of grid points used in the calculation (water points) is often the same or even smaller than the number of unused grid points (land points). It is known that domain decomposition can be used to partition the traditional one-block grid into sub-domains that reduce the unused grid points and improve performance of the ocean model [6].

Another approach, known as multiblock grid generation technique, can be used to reduce the unused grid points and to improve the performance of the model as well [5]. The multiblock grid generation technique allows choice of the grid with minimum land points along the coastline and elimination of other inland points not directly involved with the computation. The Multiblock Grid Princeton Ocean Model (MGPOM) [5] ocean circulation code has used this technique successfully. A parallel version of MGPOM assigns grid blocks to unique MPI processes with appropriate message passing commands used to share overlapping grid points.

Since not all grid blocks are of the same size, the workload varies between MPI processes. Multithreading of the computations was seen as a means of correcting this load imbalance. Two models are readily available on a variety of high performance computing platforms: OpenMP and Pthreads. But which should be used?

In this study we compare and contrast the two threading models as a programmer would. It is not our intention to differentiate OpenMP and Pthreads at a low level of implementation. Rather, a comparison by the amount of execution time reduction achievable and (perhaps more importantly) how easy each model's programming constructs are incorporated within an existing code shall be our focus.

The MGPOM code is written in Fortran. The Fortran interface of OpenMP is well-defined; however, Pthreads is only defined with a C language interface. Thus, we make use of the Fortran API to Pthreads (FPTHRD) [2] developed at the U.S. Army Engineer Research and Development Center (ERDC) Major Shared Resource Center (MSRC). The necessity of using FPTHRD adds a layer of indirection between an application code and the Pthreads functions. This reinforces the use of a higher level analysis of the threading models.

An overview of the MGPOM code using MPI is given in Section 2 with a brief review of OpenMP and the FPTHRD package given in Section 3. The process used to modify the MGPOM code for concurrent computations within MPI processes by both OpenMP and FPTHRD is discussed in Section 4. Performance results of the parallel codes (MPI-Only, MPI/OpenMP, and MPI/Pthreads) are compared in Section 4. Conclusions from this study are presented in Section 6.

2 MGPOM Code Details

MGPOM is a Fortran code that was initially designed for serial computers and later ported to vector machines. The model primitive equations describe the velocity, surface elevation, salinity, and temperature fields in the ocean. The ocean is assumed to be hydrostatic and incompressible (Boussinesq approximation).

In the parallel MGPOM version, each rectangular block grid must exchange data at overlapping grid points along the interfaces (west, south, east, and north) with adjacent blocks. Special considerations for computing with interfaces bordering open ocean, or those without adjacent blocks, are built into the model. Asynchronous receives are posted for each block interface after each set of overlap data is sent. After all data have been sent, each block processes the actual receipt of data. The exchange between blocks is completed after the data have arrived in the block's processor and been moved into the appropriate overlapping grid points.

Since all blocks synchronize to some extent at the communication routines, those blocks with less computation to be done prior to communication will tend to spend more time waiting for completion of data exchanges than blocks assigned more grid points. One possible means to balance the workload more evenly would require a complete restructuring of the grid blocks in the data set. Another method for creating a more balanced execution time between communication updates would be to further parallelize the computations in those blocks that have been assigned larger numbers of grid points.

3 OpenMP and FPTHRD

3.1 OpenMP

OpenMP [7] is a collection of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism. Directives denote parallel regions. Within parallel regions, loop iterations and/or separate blocks of code (sections) may be designated as executable in parallel. Compiler directives are also available to control access to code segments or critical regions updating shared data. Library routines and environment variables are used to control execution characteristics of OpenMP codes.

3.2 FPTHRD

Pthreads is the library of POSIX standard functions for concurrent, multithreaded programming. The POSIX standard [3] only defines an application programming interface (API) to the C programming language, not to Fortran. The FPTHRD package consists of a Fortran module and file of C routines. The module defines Fortran derived types, parameters, interfaces, and routines to allow Fortran programmers to use Pthread routines. The C functions provide the interface from Fortran subroutine calls and map parameters into the corresponding POSIX routines and function arguments.

The names of the FPTHRD routines are derived from the Pthreads root names. The string **fpthrd_** replaces the prefix **pthread_** in order to comply with the limits on subroutine names in Fortran. For consistency, all POSIX data types and defined constants prefixed with **pthread_** (**PTHREAD_**) are defined with the prefix **fpthrd_** (**FPTHRD_**) within the Fortran module.

The Fortran API preserves the order of the arguments of the C functions and provides the C function value as the final argument. This trailing integer argument is most often used to return an indicator of the termination status of the routine. Fortran interface blocks also make it possible for the status parameter to be optional in all but one Fortran routine call. One particular value of note is the FPTHRD-defined parameter **NULL** passed from Fortran to C routines. This integer is used as a signal within the C wrapper code to substitute a `NULL` pointer for the corresponding function argument.

4 Threading MGPOM

The structure of the MGPOM code contains one major loop within the main program. This loop iterates over the time-steps of the execution simulation. Within this loop are several subroutine calls (four of these are for MPI communication, while the others perform computation). The computation subroutines are composed of multiple doubly and triply nested loops operating on arrays located in `COMMON` blocks. Since most of these arrays are related to the grid blocks, they have the same dimensionality (at least within the first two defined indices).

Profiling the serial code revealed several routines that accounted for more than half of the total execution time. Threading efforts centered on these routines within the MPI version of MGPOM.

4.1 Finding Data Dependencies

Each loop within a chosen subroutine was examined in order to determine if and where OpenMP directives could be inserted. The goal was to place directives at the outermost nesting level as possible. Consider the two loops shown in Figure 1. The first loop (315) has no iteration dependencies. Thus, the OpenMP **parallel do** directive can be placed outside of the `K` loop. The second loop (230) does have a dependence (i.e., between `QF(I,J,K-1)`, `QF(I,J,K)`, and `QF(I,J,K+1)`) that requires the iterations of the `K` loop be done in order. The **parallel do** directive must be placed within the `K` loop.

Inter-loop data dependencies were checked after locating all intra-loop data dependencies

Figure 1 MGPOM Code Example for Data Dependencies

```
      DO 315 K=2,KBM1
      DO 315 J=1,JM
      DO 315 I=1,IM
        A(I,J,K)=A(I,J,K)
     &         -.50*(AAM(I,J,K)+AAM(I-1,J,K))*(H(I,J)+H(I-1,J))
     &         *(QB(I,J,K)-QB(I-1,J,K))*DUM(I,J)/(DX(I,J)+DX(I-1,J))
        C(I,J,K)=C(I,J,K)
     &         -.50*(AAM(I,J,K)+AAM(I,J-1,K))*(H(I,J)+H(I,J-1))
     &         *(QB(I,J,K)-QB(I,J-1,K))*DVM(I,J)/(DY(I,J)+DY(I,J-1))
        A(I,J,K)=.50*(DY(I,J)+DY(I-1,J))*A(I,J,K)
        C(I,J,K)=.50*(DX(I,J)+DX(I,J-1))*C(I,J,K)
  315 CONTINUE

      DO 230 K=2,KBM1
      DO 230 J=1,JM
      DO 230 I=1,IM
         QF(I,J,K)=(W(I,J,K-1)*QF(I,J,K-1)
     &           -W(I,J,K+1)*QF(I,J,K+1))/(DZ(K)+DZ(K-1))*ART(I,J)
     &           +A(I+1,J,K)-A(I,J,K)+C(I,J+1,K)-C(I,J,K)
         QF(I,J,K)=((H(I,J)+ETB(I,J))*ART(I,J)*
     &            QB(I,J,K)-DT2*QF(I,J,K))/((H(I,J)+ETF(I,J))*ART(I,J))
  230 CONTINUE
```

of the chosen MGPOM subroutines. In this case, potential read/write conflicts were looked for; that is, threads that access some array element that is updated within a previous loop that may be concurrently executed by a different thread. For such a conflict, the order of execution between the reading of an array value and the update of that array value must be done in the correct order. Such dependencies necessitate a synchronization between loops to guarantee correct execution sequencing.

In order to find any inter-loop dependencies between threads, we compiled a listing of the *read set* and *write set* of each loop. For the purposes with MGPOM, the read set of a loop is the set of all array elements that are used on the right-hand side of an assignment statement (the value is "read" from memory), while the write set is the set of all array elements that are used on the left-hand side of an assignment statement (the value is "written" into memory). After all the read/write set data have been gathered for each loop of a subroutine, the write set of each loop is compared to the read sets of all loops for any overlap.

As a concrete example of this process, consider the code extract of two loops from a subroutine of the MGPOM program shown in Figure 1. Figure 2 displays details of the write and read sets for these loops. Examination of the data in Figure 2 shows that a potential dependence exists, specifically, `A(I+1,J,K)`, `A(I,J,K)`, `C(I,J+1,K)`, and `C(I,J,K)` of the read set for loop 230 overlap with `A(I,J,K)` and `C(I,J,K)` of the write set of loop 315.

4.2 OpenMP Considerations

As stated above, intra-loop data dependencies determine the placement of OpenMP **parallel do** directives. Since all OpenMP threads synchronize automatically at the end of

Figure 2 Write Set and Read Set Form for Loops 315 and 230

Loop	**Write Set**	**Read Set**
315	`A(I,J,K)` `C(I,J,K)`	`A(I,J,K) AAM(I,J,K) AAM(I-1,J,K) H(I,J) H(I-1,J) QB(I,J,K) QB(I-1,J,K) DUM(I,J) DX(I,J) DX(I-1,J) C(I,J,K) AAM(I,J-1,K) H(I,J-1) DVM(I,J) DY(I,J) DY(I,J-1) DY(I-1,J) DX(I,J-1)`
230	`QF(I,J,K)`	`W(I,J,K-1) QF(I,J,K-1) W(I,J,K+1) QF(I,J,K+1) DZ(K) DZ(K-1) ART(I,J)` $\underline{\texttt{A(I+1,J,K)}}$ $\underline{\texttt{A(I,J,K)}}$ $\underline{\texttt{C(I,J+1,K)}}$ $\underline{\texttt{C(I,J,K)}}$ `H(I,J) ETB(I,J) QB(I,J,K) DT2 QF(I,J,K) ETF(I,J)`

parallel do constructs, it is not necessary to consider inter-loop dependencies. However, in order to avoid the overhead of creating threads multiple times within the same subroutine execution, it is possible to designate the entire subroutine be placed within an OpenMP parallel region and use the **nowait** option at the end of each loop construct to foil the automatic synchronization. Under this technique, the programmer must be sure that the **nowait** is not used between loops that have been found to contain inter-loop dependencies.

¿From previous experience [4] with the dynamic threading feature of OpenMP and the similar load imbalance characteristics inherent in the multiblock grid structure, we employed a method of using OpenMP with a dynamic number of threads per MPI process. This method involved manual control of the number of threads spawned by each process. A threshold of the minimum amount of work needed to spawn a thread is set, and each process computes the number of threads (up to some set maximum number) that should be used based on the assigned workload. The OpenMP routine OMP_SET_NUM_THREADS is called after the number of threads for the process has been computed. When a process determines that a single thread is to be used, the original serial routine is called; otherwise, a version with added directives is called. This avoids the overhead needed to create a single

OpenMP thread, perhaps multiple times within a single routine.

4.3 FPTHRD

Unlike OpenMP loop-level directives, Pthreads supports concurrency at the task or functional level. Thus, if entire subroutines from the code could be used for targets of thread creation, only minor modifications of the code would be necessary. Otherwise, efforts to thread an existing code would be centered on locating those parts of the code that could be executed concurrently and extracting the lines from the program that implement these portions into utility subroutines that are then used for thread creation and execution. It should be obvious that the former situation is more desirable.

Once candidate subroutines were identified, each was examined in more detail to ascertain whether or not the subroutine could be run concurrently. The analysis from OpenMP insertion was used for this part of the conversion. As with OpenMP, the potential for concurrency is dependent on how the arrays are partitioned and assigned to threads.

For the MGPOM code, a static allocation model was used since all arrays accessed within all loops of the chosen subroutines had the same dimensionality within the first two defined indices. Taking our cue from the OpenMP implementation, code was written to determine the number of threads to be created within the MPI process based on the size of the data block assigned to the process. Unlike OpenMP, a one-dimensional decomposition along one or the other of the first two indices or a two-dimensional decomposition of the first two indices of the block (and consequently all other pertinent arrays) into sub-blocks was then computed: one sub-block per thread to be created. The exact decomposition depended upon the number of threads assigned to the block and the relative lengths of the sides of the block. A decomposition that would yield the "squarest" sub-blocks was our goal in this study. The index boundaries of the sub-blocks within the block are saved into a global array. These index values are used by each thread created as loop iteration bounds within each threaded subroutine.

The decomposition of data determines where data dependencies can occur. Potential dependencies between threads under this static decomposition are possible when one thread accesses an array element outside the assigned sub-block; i.e., an array reference within a loop that contains an index of `I+1`, `J+1`, `I-1`, or `J-1`. Should a read (write) set contain a potential overlap index of an array contained within a write (read) set, there exists the potential for a read/write conflict, and the order of execution between these loops must be preserved for correct execution.

If loops are separated from one another by several other loops or intervening lines of code, it might be assumed that the correct execution order will naturally occur. This is not necessarily the case. The order of execution for concurrent threads is nondeterministic, and the actual execution order between threads cannot be predicted. Good programming practice requires that even when the slightest potential for some conflict to occur is present, steps must be taken to specifically ensure a correct execution ordering.

There are several methods available within the functionality of Pthreads, or that can be constructed with Pthreads routines, to coordinate execution between threads. In order to preserve simplicity within the threaded MGPOM code, a *barrier* was placed between those loops that had potential for read/write conflicts. The barrier code used is a Fortran version of the code written in C found in [1].

After identifying and inserting code to handle dependencies within subroutines, the other major chore that needs to be completed is the insertion of code to create threads that will execute the threaded subroutines. The thread creation routine allows a single argument to be sent to the subroutine. If the original subroutine that is to be threaded uses more than a single parameter, some adjustments need to be made. It is recommended that all parameters to the target subroutine be placed within a global module that can be USE-associated within the subroutine and the calling routine. This single parameter may then be an integer variable sent to the subroutine that would contain a unique thread number (within the process). Within the MGPOM code, this unique thread number is used to index the global index array for the loop bounds computed via the data decomposition subroutine.

The above is easily applied to subroutines that are called at a single point within the overall code. However, it is common practice to employ a subroutine several times within a code for performing the same computations on different parameter sets. In order to thread such a subroutine, a more involved code transformation is needed. In this instance, as before, all parameters should be encapsulated within a module for thread access. Where the subroutine header was modified above, a number of dummy subroutines are written that accept a single parameter. It is these dummy subroutines that are used in thread creation, and their only function is to call the target subroutine with the appropriate set of parameters.

Agglomerating any number of consecutive threaded subroutines can be done within a single dummy subroutine. There is no need for the subroutines to be the same. However, because some threads may complete execution of one call to a subroutine before others, the programmer must ensure that there are no data dependencies between the routines called within the dummy subroutine. A barrier call between subroutine calls would delay execution of a subsequent routine until all threads had completed execution of the prior subroutine.

5 Results

In this section, we review the code modifications needed to enhance the MGPOM code with OpenMP and FPTHRD. The ease of which these modifications were made is evaluated. Also, the performance of the MGPOM code with threaded execution is presented.

5.1 Code Modification Results

Whether using OpenMP or Pthreads, the same analysis for data dependence must be conducted. For OpenMP this analysis determines whether or not a loop can be parallelized and at what level nested loops may have directives inserted. For Pthreads the analysis is also used to find if loop iterations may be run concurrently; however, the data decomposition between threads is a factor that must be taken into account. Also, synchronization of threads between loops must also be explicitly added for inter-loop data dependencies. With OpenMP, such synchronization is the default behavior of **parallel do** loops unless otherwise specified with the **nowait** clause.

Non-loop code between loops can be ignored under OpenMP if such code is not contained within a parallel region or bracketed with a single or master directive if the entire subroutine is contained within a single parallel region. Ensuring that only a single thread executes such non-loop code must be explicitly dealt with when using Pthreads since the entire subroutine is threaded. Techniques to accomplish this task are more involved than adding a simple directive.

Besides the explicit control of threads in the face of data dependencies and non-loop code, programming with Pthreads requires the insertion of routine calls that create and join threads. Creation of new subroutines to encapsulate concurrent tasks or modification of parameter lists of existing subroutines may also be necessary. While such tasks are straightforward, they can be time-consuming. Drastic alterations to the original code may be detrimental to future code maintenance efforts. OpenMP does not force such coding additions or changes.

With the current state of OpenMP-enhanced compilers, Pthreads offers a more flexible model since a programmer is able to define and execute nested concurrency. A wider range of dynamic execution possibilities (e.g., recursive bisection, Master-Worker) are also possible with Pthreads than with OpenMP. However, if threaded execution is not desired for future executions, it is easy to recompile a program without recognizing the OpenMP directives. Removing calls to Pthreads routines, or maintaining two separate versions of the code, would be more difficult.

Given a programmer skilled in one model or the other, modifications to an existing code with OpenMP or Pthreads should present no great difficulty. More code modifications and additions are needed with Pthreads, though. However, we feel that such additional work should not preclude use of Pthreads. Thus, from a purely programming point of view, the choice of which model to use may depend upon the skills of the programmer. Exceptions to this would be a case where few loops, placed sporadically throughout the code, are to be parallelized would tend to favor OpenMP, while the case of a code containing various independent tasks with differing amounts of work would favor Pthreads.

Table 1 Twenty Block Grid Overall Execution Times (in minutes)

MPI Only	MPI/OpenMP	MPI/Pthreads
69	34	48

5.2 Performance Results

The Persian Gulf is the physical geographic area selected for this study to demonstrate the execution performance of the different threading models. This area extends from 48° East to 58° East in longitude and from 23.5° North to 30.5° North in latitude. Part of the Gulf of Oman is also included in this physical domain. The complicated features of the coastlines near Qatar, along the Strait of Hormuz, and the northern part of the Persian Gulf is a good data set to demonstrate the advantages of multiblock grids within the MGPOM code.

We present the overall performance results for a 10-day simulation on the twenty-block grid in this section. The simulations were computed on the SGI Origin 2000 located at ERDC MSRC. Only 20 processors were used in the MPI-Only version. For threaded runs of the code, the maximum number of threads per process was set to 4. Based on the grid points per thread threshold that was decided upon and the sizes of the grid blocks, 44 processors were made available for the OpenMP and FPTHRD threaded runs.

Table 1 shows the timing results from the MPI-Only, MPI/OpenMP, and MPI/Pthreads versions of the MGPOM code on the twenty-block grid. The threaded results are based on modifying only four routines from the MGPOM code. These were the four routines that accounted for the most total execution time within the serial version of the code.

5.2.1 Idle Overhead Reduction

While MGPOM processes use asynchronous communication, those processes must synchronize to some degree; e.g., processes with small blocks are forced to wait on the actual receipt of data from processes with large blocks that have been performing more calculations prior to communication. The greater the difference in size between adjacent blocks, the larger the load imbalance of computation will be. In order to quantify the degree of load imbalance within a given code segment, *idle overhead* is defined as the ratio of execution time to maximum execution time expressed as a percentage:

$$\text{idle overhead} = 100\% \times \left(1. - \frac{\sum_{i=1}^{N} t_i}{N \times t_{max}}\right) \tag{1}$$

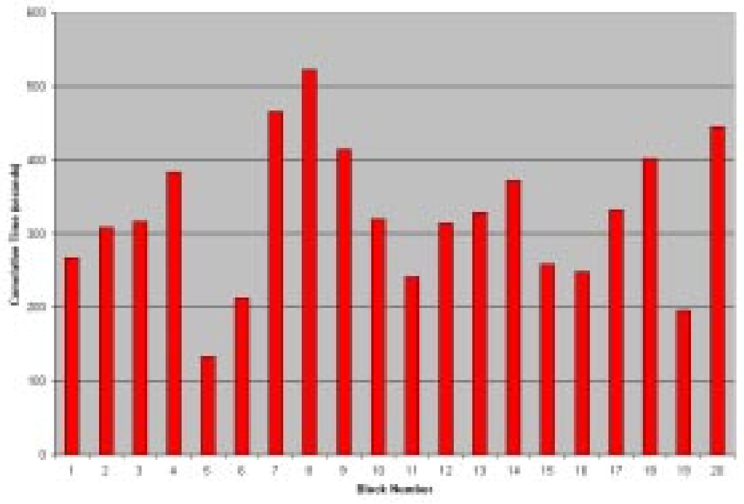
where N is the number of MPI processes, t_i is execution time of process i, and t_{max} is the largest execution time of a process.

Profiling the serial MGPOM code revealed the routine `PROFQ` takes nearly 20% of the total execution time. The load balancing effects of OpenMP and Pthreads on this routine are examined. The following results were taken from the executions reported above.

Figure 3 shows the cumulative timing results of the `PROFQ` routine for individual blocks of the twenty-block, MPI-Only code for a 10-day simulation running on 20 SGI Origin 2000 processors. A measure of 38% idle overhead was observed within this routine (total wall-clock execution time was 69 minutes).

With the MPI/OpenMP version of the code, a measure of 21% idle overhead was observed (total wall-clock execution time was 34 minutes). Figure 4 shows the `PROFQ` routine timing results for this version of the code. Under the MPI/Pthreads code, a measure of

Figure 3 PROFQ (MPI-Only) Cumulative Execution Time in seconds



28.5% idle overhead was achieved within the `PROFQ` routine (total wall-clock execution time was 48 minutes). This timing result is shown in Figure 5.

6 Conclusions

The timing results show a significant improvement in the execution time as well as in the load imbalance produced by MPI-Only execution. It has also shown that through use of the hand-coded, dynamic threading within OpenMP and Pthreads, load balance between the MPI processes can be improved. The techniques described herein should be applicable to a large number of other scientific codes.

Since only one specific code was used in this study, a definitive conclusion about which thread programming model is superior cannot be drawn. However, within the constraints of the limited amount of modifications made to the MGPOM code, OpenMP appears to be the better choice. It was easier to program with and delivered a faster execution time.

Figure 4 PROFQ (MPI/OpenMP) Cumulative Execution Time in seconds

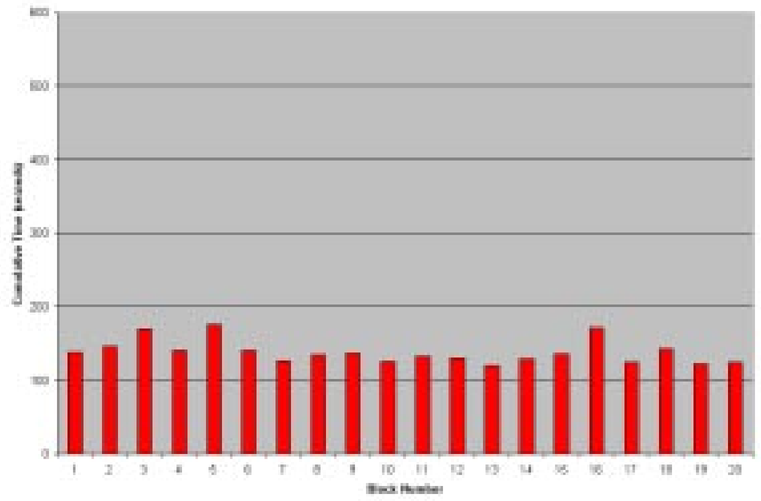
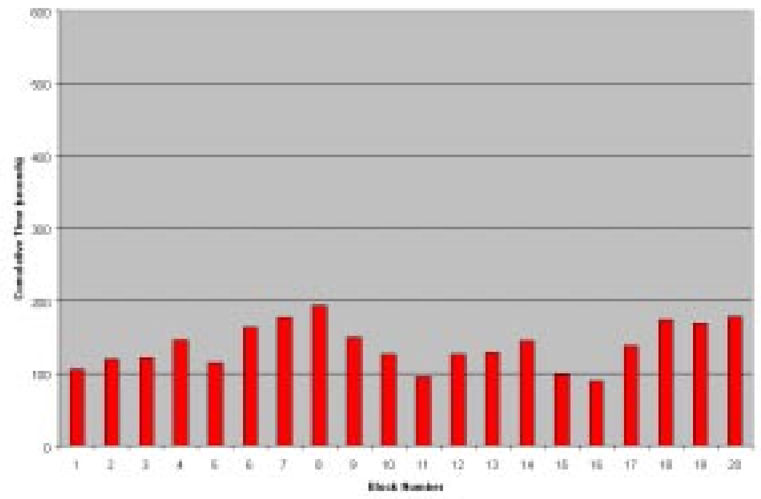


Figure 5 PROFQ (MPI/Pthreads) Cumulative Execution Time in seconds



Whether or not having an added layer of subroutine calls required with the use of the FPTHRD package was detrimental to that version of the code can only be deduced by further study. Also, the question of whether similar finding would result from a more complexly structured code must await future research.

References

[1] David R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, Reading, MA, 1997.

[2] Richard J. Hanson, Clay P. Breshears and Henry A. Gabb. "A Fortran Interface to POSIX Threads," Technical Report ERDC MSRC/PET TR/00-18, February 2000.

[3] *9945-1:1996 (ISO/IEC) [IEEE/ANSI Std 1003.1 1996 Edition] Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface (API) [C Language] (ANSI)*, IEEE Standards Press, 1996.

[4] Phu V. Luong, Clay P. Breshears, and Henry A. Gabb, "Dual-level Parallelism Improves Load-Balance in the Production Engineering Application CH3D," Technical Report ERDC MSRC/PET TR/00-07, February 2000.

[5] Le N. Ly and Phu V. Luong, "Numerical Multiblock Grids in Coastal Ocean Circulation Modeling," *Journal of Applied Mathematical Modeling*, **23**, pp. 865–879, November 1999.

[6] Wade D. Oberpriller, Aaron C. Sawdey, Matthew T. O'Keefe, Shaobai Gao, and Steve A. Piacsek, "Parallelizing the Princeton Ocean Model Using TOPAZ," `http://topaz.lcse.umn.edu`.

[7] OpenMP Architecture Review Board, "OpenMP Fortran application program interface, Version 1.1," `http://www.openmp.org`, November 1999.

[8] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra, *MPI—The Complete Reference: Volume 1, the MPI Core*, MIT Press, Cambridge, 1998.